AD-A113 723 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
AUTOMATIC CONSTRUCTION OF SPECIAL PURPOSE PROGRAMS,(U)
JAN 82 C GOAD MDA903-80-C-0102
UNCLASSIFIED STAN-CS-82-897 NL

END
DATE FILMED
5-82
DTIC

January 1982

Also numbered:
AIM-344

Report. No. STAN-CS-82-897

Automatic Construction of Special Purpose Programs

by

Chris Goad

Department of Computer Science

Stanford University
Stanford, CA 94305

Automatic Construction of Special Purpose Programs

Chris Goad
Computer Science Department
Stanford University

ABSTRACT

According to the usual formulation of the automatic programming task, one starts with a specification of a programming problem, and seeks to automatically construct a program satisfying that specification. This paper concerns a different style of automatic programming. Rather than defining the class of programming problems to be dealt with by the language in which those problems are formulated, we instead consider classes of problems defined in ordinary mathematical terms. Also, our aims are different from the traditional aims of automatic programming in that we are interested primarily in increasing the efficiency of computations, rather than in transfering the burden of programming from human to computer. Let $\alpha(p, x, y)$ be a ternary predicate. Suppose that in the course of some large computation we are obliged to repeatedly compute values of y with $\alpha(p, x, y)$ from given values of p and x. Suppose further that in the sequence of p's and x's to be treated, p changes slowly and x rapidly. Then we seek to automatically synthesize a fast special purpose program A_p for each p; A_p is expected to compute a y with $\alpha(p, x, y)$ when given x as input. We present one example of special purpose automatic programming in detail, namely, a method for synthesizing special purpose programs for eliminating the hidden surfaces from displays of three dimensional scenes. (Hidden surface elimination is one of the central problems in three dimensional computer graphics). In a test of the method, a synthetic program specialized to treating a particular scene - but from an arbitrary point of view - proved to be an order of magnitude faster than the best available general purpose algorithm.

This research was supported in part by the National Science Foundation under Grant MCS81-04873 and in part by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-80-C-0102

Automatic Construction of Special Purpose Programs

1. Introduction

The automatic programming problem as traditionally formulated is that of passing automatically from specifications of programs (expressed in some particular formal language) to programs which satisfy those specifications. We will be concerned here with a different style of automatic programming. Rather than defining the class of programming problems to be dealt with by the language in which those problems are formulated, we will instead consider smaller and more manageable classes of programming problems; classes of problems defined in ordinary mathematical terms. Also, our aims are different from the traditional aims of automatic programming in that we are interested primarily in increasing the efficiency of computations, rather than in transfering the burden of programming from human to computer. (There is of course some overlap between these aims.)

We will introduce the kind of automatic programming which we have in mind by means of an example - an example which will be treated at length later in the paper. The example concerns an important computational problem from three dimensional graphics, namely, the hidden surface elimination problem. The hidden surface elimination problem is that of determining, given a formal description of one or more opaque objects in three dimensional space, which surfaces of those objects are visible from a given point of view, and which are hidden behind other objects in the scene. In many applications of three dimensional computer graphics, such as computer animation and flight simulation, the appearance of the same scene must be computed repeatedly for many different positions of the viewer. Several algorithms for hidden surface elimination have been devised which exploit this property of an application (which is conventionally referred to as "object coherence"). In each case, the algorithms proceed by first constructing a suitable data structure describing the scene; this is done "off-line". Then the data structure is used at runtime for performing the hidden surface computation for each position of the viewer.

Another way of exploiting this kind of coherence is to automatically construct a different special purpose program for each scene treated. This scheme involves devising an automatic *synthesis method* M : M is a program which takes a scene s as input, and yields a special purpose program Q_s as output. Q_s in turn takes the position of the viewer as input and produces a suitable representation of the scene with hidden surfaces removed as output. Since Q_s has a very limited task to perform, it is potentially much faster than any general purpose algorithm for hidden surface elimination would be for the same scene. We have applied the automatic programming scheme to hidden surface elimination, with good results. The special purpose programs which we have produced are indeed much faster than any general purpose algorithm to be found in the computer graphics literature.

The hidden surface example illustrates a very general method for improving the efficiency of computation by use of automatic programming. The essential property of the example from the standpoint of automatic programmming is this: a computation taking two inputs (here, the scene s and the viewer position p) is performed repeatedly in a context where the first input changes only occasionally whereas the second input changes rapidly. Any situation which exhibits this property is a candidate for special purpose automatic programming; we may attempt to develop a synthesis method which takes the slowly changing input and generates a fast special purpose program; the special purpose program in turn takes the second input and completes the computation. An overall

gain in efficiency is realized if the time saved by the use of the special purpose programs on the rapidly changing input exceeds the time lost in the occasional generation of a new special purpose program upon presentation of a new value of the slowly changing input.

Let us restate the points of the last paragraph in more formal terms. Let $\alpha(p, x, y)$ be a ternary predicate. Suppose that in the course of some large computation we are obliged to repeatedly compute values of y with $\alpha(p, x, y)$ from given values of p and x, and that in the sequence of p's and x's to be treated, p changes slowly, and x rapidly. The automatic programming problem for α is just that of automatically passing from any value for p to a program for $\lambda x\, y.\alpha(p, x, y)$. That is, we want a program S_α such that for each p, and each x, $\alpha(p, x, S_\alpha(p)(x))$ holds. The utility of a solution to the automatic programming problem defined by α depends on the factors mentioned in the last paragraph.

In the situation described above, where y with $\alpha(p, x, y)$ is to be computed from slowly changing p and rapidly changing x, we will say the sequence of inputs exhibits ***sweep coherence***. (The picture behind this term is that of a ray performing a circular sweep about a point, with x "attached" farther out than p, so that the position of x changes more rapidly than that of p. Evidently, object coherence in three dimensional graphics is an instance of sweep coherence.) There is a conventional scheme for exploiting sweep coherence which was alluded to earlier in connection with hidden surface elimination. Namely, one devises a data structure which encodes the relevant information about p in such a way that the computation of y can be completed rapidly once a value of x is given. To take the archetypical computer science example, if one needs to repeatedly determine whether a number x belongs to a list p of numbers, for many values of x but only a few of p, one proceeds by sorting each new p, and then using binary search to determine whether each successive x belongs to the current p. Note that there is no convincing way to ***formally*** distinguish between the "data structure scheme" and the "automatic programming scheme". The reason for this is that automatically constructed special purpose programs may be regarded as data structures which, together with their arguments, are passed to the interpreter for the language in which the programs are expressed. However, the fact that the distinction between a program and a data structure which is not a program cannot be convincely formalized does not make it a useless distinction. For the purposes of this paper, the word "program" is to be read in its ordinary informal sense: a program is a description of a method of computation which is expressible in a programming language of the usual kind.

The principal purpose of this paper is to present evidence that special purpose automatic programming is a useful enterprise - or, more precisely, to present new evidence, since some such evidence exists already in the computer science literature (see section 6). Of necessity, individual concrete examples are the only kind of evidence which can be expected, since at the outset we are supposing that for each programming task (given by a ternary predicate α) a different synthesis program S_α will be required. As already indicated, we will consider one example in detail, namely, the hidden surface elimination problem.

Another purpose of the paper is to indicate that although different synthesis programs are needed for different automatic programming problems, there are nonetheless general - and, in fact, quite primitive - techniques which can be used for a variety of problems. These generally useful techniques are for the most part already in use in automatic programming and automatic program transformation of the traditional kind. Our example shows that these general techniques, although not always very effective alone, can produce good results when combined with considerations which are particular to the (narrow) class of programming problems treated in an instance of special

purpose automatic programming - that is, particular to the mathematical specification which we have designated α.

Before closing the introduction, we wish to emphasize the following point. The character of the problems which are treated in special purpose automatic programming is very different from the character of those dealt with in automatic programming of the traditional kind. The traditional style of automatic programming work deals with universal or metamathematically defined classes of problems in that the collection of problems to be solved by a real or ideal automatic programming system is defined by giving the language in which the problems are to be formulated, rather than by reference to particular objects and operations to be carried out on those objects. In computing practice as a whole, this metamathematical or universal style of defining the class of problems to be solved by a particular program is very unusual. The reason for this is simply that such methods as have been found for attacking universal classes of problems are rarely efficient enough to compete with special purpose methods designed for the particular situations which come up in practice. One prefers to solve broad classes of problems at one blow if possible, but such possibilities do not often arise.

In special purpose automatic programming, on the other hand, the class of problems to be solved by any particular synthesis program is defined by an ordinary mathematical predicate α which *will* in general make reference to specific objects and operations. In this sense, the style of automatic programming with which we are concerned is much closer to ordinary computing practice - and much farther away from the universal aims popular in artifical intelligence - than the traditional style. All of this shows, in any case, that automatic programming - in the sense of writing programs which write programs - is not intrinsically bound to universal aims.

For a general discussion concerning choices of problem classes in computing and mathematics, see Kreisel[1981].

2. A simple example of special purpose automatic programming: list membership

The method which we use for synthesis of special purpose hidden surface elimination programs was developed and will be presented in several stages. We will start by describing a simple scheme for automatically generating special purpose programs, namely the scheme of specializing general purpose programs. If this scheme is applied in a direct way to the hidden surface problem, the results are unsatisfactory in that the special purpose programs P_s which are generated for any but very small scenes s are intractably large. However, having formed a picture of what the programs P_s are like, we are able analyse their defects. The result of this analysis is a series of modifications to the P_s which finally yield usable programs Q_s. More precisely, the modifications apply not to the synthesized programs P_s, but to the method by which they are synthesized, so that we end up with a direct method for synthesis of usable special purpose programs. We have chosen to present the synthesis method in a step-by-step manner in order to increase clarity, but also for a more important reason. Namely, the method by which the P_s are produced is directly applicable to a wide variety of problems, whereas the subsequent modifications are particular to the hidden surface problem. Thus the multi-stage presentation has the virtue of exhibiting the method which we use for the hidden surface problem as an instance of a general scheme for exploiting sweep coherence; the scheme is: "take a general purpose algorithm and specialize it, analyse the results, and introduce appropriate modifications."

The general side of our presentation, that is, the side concerned with the specialization of programs, will begin with an example which lacks the technical complexities of the hidden surface

problem, but which illustrates several techniques which are used later. The example is essentially a reformulation of the construction of a binary search tree by random insertion [Robson, 1979]. Let u be a list of elements from a linearly ordered set, and let n be an individual member of that set. The computational problem which we wish to solve is that of determining whether n is a member of u. We may set this up as a problem of the kind described in the introduction by defining $M(u,n,r)$ as "r = TRUE if $n \in u$; r = FALSE otherwise". We assume that the sweep coherence criterion for the utility of performing a computation in two stages is met; that is, we assume that membership of n in u must be determined for many different values of n, but few values of u. What is wanted is a synthesis program for M - a program S_M which will take a list u as input and produce as output a fast program $S_M(u)$ for determining membership in that list.

The appropriate "data stucture scheme" for exploiting sweep coherence in the current example involves sorting and binary search. Since this method is provably optimal, no improvements by the use of automatic programming are possible in this case.

With this warning in mind, let us proceed. The synthesis method for M consists of taking a slightly peculiar algorithm for computing list membership and specializing it to the list at hand. The algorithm to be specialized is given by the recursive definition:

$f(u,n)$ = if $empty(u)$ then FALSE else
if $n < head(u)$ then $f(tail(u),n)$ else
if $n > head(u)$ then $f(tail(u),n)$ else
TRUE

The specialization to a concrete list L is carried out by a conventional scheme: first symbolically execute f as applied to L, then simplify the resulting decision tree by removing redundant decision nodes. (Other terms for symbolic execution include partial evaluation [Beckeman et al, 1976], and repeated unfolding [Burstall and Darlington, 1977]). A redundant decision node is one whose outcome is predetermined by the outcomes at nodes along the path leading from the root of the decision tree to the node in question. More precisely, if $P_1 \cdots P_k$ are the decision predicates appearing on the path leading to node N, and Q is the predicate at node N, then N is redundant if $A_1 \wedge A_2 \cdots A_k \supset Q$ is valid, or if $A_1 \wedge A_2 \cdots A_k \supset \neg Q$ is valid, where $A_i = P_i$ if the true branch is taken out of P_i's node in the path leading to N, and $A_i = \neg P_i$ if the false branch is taken. The removal of a redundant decision node is carried out by replacing the subtree of which it is a root by the subtree rooted at its left or its right son, depending on whether the predicate at the node is predetermined to be true or false. Note that determining whether a node in a decision tree is redundant involves deciding whether certain formulas are valid. In the current instance, all such formulas are implications between conjunctions of inequalities containing n as the only variable, so that the decision problem has an easy automatic solution.

Here is an example of the behavior of S_M, the synthesis method for M just described, in a concrete case. Suppose that the underlying ordered set for the membership computation is the integers, and that we wish to compute $S_M(\langle 2,5\rangle)$. We proceed by symbolically executing $f(\langle 2,5\rangle, n)$, arriving at the decision tree:

```
if n < 2 then
  (if n < 5 then FALSE else
  if n > 5 then FALSE else
  TRUE) else
if n > 2 then
  (if n < 5 then FALSE else
  if n > 5 then FALSE else
  TRUE) else
TRUE
```

The decision node represented by the second line of the above program is redundant, since $n < 2 \supset n < 5$. Its removal yields:

```
if n < 2 then FALSE else
if n > 2 then
  (if n < 5 then FALSE else
  if n > 5 then FALSE else
  TRUE) else
TRUE
```

No redundant nodes remain. Hence the above program represents the final result of computing $S_M(\langle 2,5 \rangle)$. By the analysis given in [Robson 1979], the behavior of S_M exhibits the following general properties. First, the expected maximum depth of the decision tree generated by S_M when applied to a list L of length n (assuming that L is given in random order with equal probability assigned to each ordering) is $k \log_2(n)$ for a constant k between 7.26 and 8.62 (the exact value of k is not known). Second, the number of decision nodes in $S_M(L)$ is exactly $2n$, where again n is the length of L. (If we adopt a formulation in which the decision nodes have three branches [for $<$, $=$, and $>$], then the number of nodes and depth are halved.) Thus the worst case running time of the programs generated by S_M, measured by the maximum number of decision nodes to be traversed on the path to the result, may be expected to exceed the optimal worst case running time obtained by sorting and binary search by only a small constant factor. Also the size of the programs generated is linear in the length of L, so that in this respect as well S_M is only a small constant factor worse than optimal. Finally, note that expected worst case running time of $S_M(L)$ is immensely better than that of the one stage algorithm given by f; the former is logarithmic in the length of L, while the latter is linear.

This example illustrates the following points.

(1) Exploiting sweep coherence is by no means a marginal matter in computer science. The sorted list is just one of many data structures devised for this purpose, but even taken alone sorting has been the target of a large amount of effort.

(2) In the case of list membership, the automatic programming approach failed to produce results which were as good as those produced by the right data structure. But on the other hand, our approach did almost as well, and in a sense required less in the way of intellectual resources than the conventional scheme. For, in constructing special purpose programs for individual lists L, all that was required was the unwinding and optimization of a slight variant of the ordinary one stage program for computing membership. On the other hand, the construction of the sort and binary search method requires additional ideas - ideas which are not already implicit in the one stage program. There are many computational problems exhibiting sweep coherence where a one stage program exists, but where no fully satisfactory data structure for exploiting the coherence

is available - the required additional ideas have not been found. Such problems are promising candidates for the automatic programming approach. One such problem, which as we will see has fulfilled its promise, is that of hidden surface elimination.

3. The synthesis method for hidden surface elimination

Now we turn to the our principal subject, the synthesis of special purpose programs for hidden surface elimination.

Recall that the hidden surface elimination problem is that of taking a scene s and a viewer position p, and computing the appearance of s from the viewpoint p with hidden surfaces removed. Many approaches to the hidden surface elimination problem have been developed [Sutherland et al,1974]. The general approach which will concern us here has the following attributes:

The scene to be displayed is represented by a set F of faces, where a face is a convex polygon with some particular position and orientation in three dimensional space. A face is oriented also in the sense that it has a front and a back side. (A face in turn may be given by the ordered list of its vertices, and the vertices by their coordinates in three space.) The faces in F fit together to form (approximations to) the surfaces of the objects in the scene. This representation scheme is currently the one most commonly used in three dimensional computer graphics.

In order to display the scene from a particular point of view, the faces are first sorted into what is called "priority order". A list L of faces is in priority order with respect to a given viewpoint p if whenever face i occludes face j from p, i appears after j in L. (A face i is said to occlude a face j from p if *some* part of j is hidden behind i as viewed from p.) Once a priority ordered list has been computed, the generation of a picture of the scene with hidden surfaces removed can be carried out by a "painting" process, in which the faces are written onto the picture (eg by writing onto the memory of a bit-mapped CRT) in the order in which they are given in L. Then, if one face partially hides another, the hiding face will be written (or painted) after the hidden face; thus elimination of hidden surfaces occurs by overwriting. The priority list approach appears to be the best available for real time applications such as flight simulation; in such applications the painting process is carried out largely by special purpose hardware. (Notes: (1) the faces which are oriented away from the viewer - that is, the "back faces" must be eliminated before the painting is done. (2) For some scenes and viewpoints, no priority order exists; see [Sutherland et al, 1974] for an example. However, this happens rarely for naturally occuring scenes. The priority sorting methods which we will consider will perform the priority sort if possible, and will indicate its impossibility otherwise.)

Let $P(F,p,L)$ denote the predicate, "L is a priority sorted list of the faces F from viewpoint p". Our aim is to devise a synthesis program for P, that is, a program S_P which will take F and generate a special purpose priority sorting program $S_P(F)$; $S_P(F)$ then takes the position p as input, and generates a priority sorted list L for p.

We will proceed by first considering the result of applying the direct approach described in section 2 - that of specializing a simple one stage algorithm and then removing redundancies from the resulting decision tree. Then modifications of the direct method will be introduced one by one until a usable final result is obtained.

Here is the brute force algorithm for priority sorting:

(1) Compute the entire occlusion relation; that is, determine for each pair of faces i, j with $i \neq j$ whether or not i occludes j from p.

(2) Topologically sort the faces according to the occlusion relation computed in step (1). If there is a cycle in the occlusion relation, then no topological sort is possible, and in this case the outcome of the computation is an indication of failure. (Priority sorting consists exactly of finding a linear order which is consistent with the occlusion relation; the task of extending an acyclic binary relation to a linear order is the topological sorting problem. Algorithms for topological sorting may be found in standard references such as Knuth[1968]).

Now, consider the decision tree which results from unwinding this algorithm for a particular set F of faces. The decision tree T_0 which we have in mind may be described in precise terms as follows. Let $n = |F|$, and let $\langle i_1, j_1 \rangle, \langle i_2, j_2 \rangle \cdots \langle i_{n(n-1)}, j_{n(n-1)} \rangle$ be an enumeration of the set of pairs of faces from F given in the order in which they are considered in step (1) of the brute force algorithm. T_0, then, is a full binary tree of depth $n(n-1)$ (with $2^{n(n-1)}$ nodes!). Let $occ(p, i, j)$ denote the occlusion predicate: $occ(p, i, j)$ holds iff i occludes j from p. Then the predicate appearing at each of the 2^{k-1} decision nodes at the kth level of the tree is $occ(p, i_k, j_k)$. Evidently, each leaf of the tree corresponds to a particular truth assignment to the occlusion predicates, that is, to a particular occlusion relation on the faces. The computational result appearing at each leaf is the result of topologically sorting the faces according to the occlusion relation associated with that leaf, or an indication of failure if that relation contains a cycle.

The next stage in the process consists of the removal of redundant nodes from the decision tree. However, this requires that we automatically decide whether assertions of the form $occ(p, i, j)$ or $\neg occ(p, i, j)$ follow from sets of other assertions of the same form. As they stand these decision problems do not have any efficient solution. This difficulty can be overcome by simplifying the predicates which appear in the decision tree in the following way. For particular faces i and j, the predicate $occ(p, i, j)$ can be expressed as a conjunction of simpler predicates $occ_1(p, i, j) \cdots occ_r(p, i, j)$. The number r of such predicates is just the sum of the number of vertices in i and the number of vertices in j. Each predicate $occ_m(p, i, j)$ has the form, " the kth vertex of j (or i) lies below (or above) the 'horizon' defined by the lth edge of i (or j), as seen from p", or, equivalently, "p lies above (or below) the plane defined by the endpoints of the lth edge of i (or j) and the kth vertex of j (or i)". Thus, each predicate $occ_m(p, i, j)$ can be written as a linear inequality in the coordinates p_x, p_y, p_z of the viewer position p, with coefficients depending on i, j. Now, let us rewrite the decision tree T_0 to get a decision tree T_1 in which the predicate at each decision node is a linear inequality. This is done by expanding out the occlusion decision nodes according to the scheme:

if $occ(p, i, j)$ then t_1 else $t_2 \rightarrow$

```
if occ1(p,i,j) then
   if occ2(p,i,j) then
      .
         .
            .
               if occr(p,i,j) then t1 else
               t2 else
            .
         .
   t2 else
t2
```

We are in a good position to automatically remove redundant nodes from T_1, since (1) the predicate at each of its nodes is a linear inequality, (2) the negation of a linear inequality is a linear inequality, and (3) the question of whether a given inequality follows from a set of inequalities can be efficiently decided by use of the simplex algorithm. Further, on intuitive grounds, it is to be expected that the removal of redundant nodes will reduce the size of T_1 by a very large factor, since, once the, say, fifth level of the decision tree has been reached along a given path, the tests met so far along the path will have severely constrained the position of the viewer, and accordingly it is likely that the outcomes at the great majority of nodes below that level will in fact be predecided.

Of course, the problem with T_1 from the practical standpoint is that, even for very small numbers of faces, its size is intractably large. In practice, however, one will not proceed by first generating the decision tree T_1, and then optimizing it, but instead will optimize the tree while it is being generated. This can be done by constructing the tree from top down. When a new decision node is to be generated, one asks first whether its outcome is predetermined. If so, then one need not add the node to the tree; instead, one proceeds directly with the generation of its left or right subtrees, depending on whether the predicate at the node is predetermined to be true or false. If this scheme is used, then no redundant nodes are ever generated; instead, the optimized tree is produced in one pass.

So, we have shown how to automatically construct an optimized decision tree T_2 for doing the priority sort for a fixed set of faces, but variable viewpoint. Although we expect that T_2 will be much faster than the brute force method from which it sprang, it is still too large to be of practical use.

The following observations will allow us to do much better. (1) The entire occlusion relation need not be determined in order to do the priority sort. (2) A partial determination of the occlusion relation which is insufficient to do the entire sort may still allow a part of the sort to be carried out. The former observation will allow us to shorten the tree, while the latter will make it possible to diminish the size of the results appearing at the leaves, by moving as much information as possible about the output of the computation to nodes closer to the root of the tree.

Let I be a set $\{I_1 \cdots I_k\}$ of linear inequalities in the the coordinates p_x, p_y, p_z of the viewer position. We may think of I as representing the simplex of points in three dimensional space which satisfy all of the inequalities $I_1 \cdots I_k$. Let $canocc(I, i, j)$ denote the predicate, "there is some point satisfying each of the inequalities in I from which the face i occludes the face j". Now, consider an arbitrary node N in T_2. Let I_N be the set of inequalities which are assumed to hold at N - that is to say, the set of inequalities which must hold if N is to be reached in the course of executing the decision tree. Let G be the graph of $canocc(I_N, i, j)$ viewed as a binary relation on the set of faces. If G is acyclic, then a topological sort of G will yield a priority order which is valid for all view points in the simplex I_N. In this case, the subtree rooted at N is not needed at all; a correct priority order can be generated without further case analysis. Suppose, on the other hand, that G contains cycles. Let $S_1 \cdots S_k$ be the strongly connected components of G. (Recall that a strongly connected component of a directed graph is a maximal set S of points from the graph having the property that, for any two points $p, q \in S$, there is a path from p to q.) As long as there is more than one strongly connected component - that is, as long as G is not itself strongly connected - a part of the priority order can be determined at the current stage. Let G' be the (acyclic) graph which results from collapsing the strongly connected components of G into single vertices. (Formally: the vertices of G' are the strongly connected components $S_1 \cdots S_k$ of G; $[\![S_i \rightarrow S_j]\!]$ is an edge in G' iff for some $p \in S_i$, $q \in S_j$, $[\![p \rightarrow q]\!]$ is an edge in G.) The result of topologically sorting G' gives an ordering to

the $S_1 \cdots S_k$ which constitutes a partial priority sort, in the following sense. Let G'' be the graph of any occlusion relation which is consistent with inequalities I_N. If G'' has a priority ordering at all, then it has one of the form $append(p_1(S_{i_1}), p_2(S_{i_2}) \cdots p_k(S_{i_k}))$ where $S_{i_1} \cdots S_{i_k}$ is the topologically sorted ordering of G', and where $p_j(S_{i_j})$ is some permutation of S_{i_j}. Thus, the set of faces has been partitioned into subsets $S_1 \cdots S_k$ such that the members of each S_i may be listed consecutively in any final ordering, and the order in which the S_i appear has also been decided. It is only the orderings within the S_i that remain to be determined. All of this has two consequences: (1) the occlusion or lack of occlusion between faces in different strongly connected components need not be considered in the decision tree rooted at N. Further, the strongly connected components may be considered separately; if desired, a different decision tree may be generated for each. (2) The decision tree rooted at N needs to specify only the orderings within, and not between the strongly connected components, provided that the ordering $S_{i_1} \cdots S_{i_k}$ is stored in one way or another at the node N.

The observations of the last paragraph indicate in a fairly direct way how T_2 may be improved on. The result T_3 of this improvement has a structure which is a bit more complicated than that of an ordinary decision tree, in that it has internal nodes which are not decision nodes. One way of describing T_3 is as a simple loop-free program which is built up from constants denoting lists of faces by use of (1) the conditional operator: "if P then t_1 else t_2" where P is a linear inequality in p_x, p_y, p_z, and (2) the *append* operator: "$append(t_1, \cdots t_k)$". Thus, T_3 differs from a decision tree only in that T_3 makes use of two operators ("if" and "*append*"), rather than just one ("if") in constructing its result.

The method by which T_3 is built follows closely the method used to build T_2. The difference is that the *canocc* graph is employed to guide the selection of face pairs for case analysis, and also to split the computation of the priority ordering into separate computations for separate strongly connected components. The method for synthesizing T_3 is given below by the recursive program $\mathcal{R}(I, F)$. Here, I is a set of inequalities, and F a set of faces. The result returned by $\mathcal{R}(I, F)$ is a program which computes an ordering for the faces in F; under the assumption that the inequalities in I hold, this ordering will be a correct priority ordering of the faces. $\mathcal{R}$ makes use of a subroutine $\mathcal{R}_1$ which takes care of generating the individual tests $occ_1(i,j) \cdots occ_r(i,j)$ which together determine whether i occludes j. $\mathcal{R}$ and $\mathcal{R}_1$, then, are as follows: (For the sake of clarity, we make the simplifying assumption here that the set of faces F has a priority ordering from all viewpoints; the other case is not difficult to handle.)

$\mathcal{R}(I, F)$:

(1) Compute the graph G of $canocc(I, i, j)$ for $i \neq j \in F$.
(2) If G is acyclic, then topologically sort G, and return the constant representing this sorted list of faces.
If G is strongly connected then:
(a) choose i, j such that $occ(i, j)$ is not decided by I.
(b) return $\mathcal{R}_1(I, F, i, j, 1)$ (This does a case analysis according to whether i occludes j)
Otherwise:
(a) Let $S_1 \cdots S_k$ be the strongly connected components of G. Topologically sort the graph G' gotten by collapsing the S_j, getting an ordering $S_{i_1} \cdots S_{i_k}$ of the S_j. Compute the programs $P_j = \mathcal{R}(I, S_j)$ for priority sorting the S_j. Return the program, "append"$(P_{i_1}, \cdots P_{i_k})$

$\mathcal{R}_1(I, F, i, j, n)$:

(Here $\langle i, j \rangle$ is the pair of faces to be dealt with; n is the index of the occlusion test to be generated)

(1) If n is greater than the number of occlusion tests needed for the faces i, j - that is, if all the tests have already been generated - then return $\mathcal{R}(I, F)$

Otherwise:

(a) If I implies that $occ_n(i, j)$ holds then return $\mathcal{R}_1(I, F, i, j, n+1)$.

(b) If I implies that $occ_n(i, j)$ does not hold then return $\mathcal{R}(I, F)$.

(c) Otherwise, generate the test; return the program:

"if" $occ_n(i, j)$ "then" $\mathcal{R}_1(I \cup \{occ_n(i, j)\}, F, i, j, n+1)$ "else" $\mathcal{R}(I \cup \{\neg occ_n(i, j)\}, F)$

4. Results of experiments

A program for synthesis of special purpose priority sorting programs has been implemented on the Stanford computer science department PDP-10/KL-10 computer in MacLisp. The program has been tested on one large scale example so far, namely, a description of a hilly landscape derived from a data base provided to the author by the Link division of Singer corporation (Link is a manufacturer of flight simulators). The description consisted of a set L of 1135 faces making up, roughly speaking, a triangulation of the landscape[1]. The implemented program is based directly on the method described in the last section, but includes the following important refinement. (There are also a number of less important refinements and implementation details whose description is beyond the scope of the current general presentation of the method).

In almost all three dimensional computer graphics applications, the field of view covered by the image to be generated is limited. For example, in several of the flight simulators manufactured by Link, the field of view or "window" spans 48 degrees horizontally, and less than 40 degrees vertically. We exploit this fact by performing an initial case analysis according to the direction in which the viewer is looking. Specifically, this is what is done: Consider the projection of the viewing direction v onto the x, y plane. We divide the "pie" consisting of the set of all possible such projections into ten equal "slices", each 36 degrees wide. Now, one face can *visibly occlude* another only if there is a ray from the viewer's eye which passes through face i and face j *and* whose direction lies within a certain angular distance the viewing direction v. So, the assumption that the x, y projection of v lies within a given 36 degree slice reduces the number of possible occlusions between faces, since it places limitations on the relative positions of visibly occluding pairs of faces. In any case, the ten slices of the pie are considered separately, with field of view parameters of 48 degrees horizontal and 40 degrees vertical, and with the additional assumption that the angle of the viewing direction to the x,y plane (that is, the angular deviation from horizontal flight) is less than 30 degrees. The other cases, where the angle of view is steeply up or steeply down, are handled separately. For each slice, a separate initial *canocc* relation is computed; it is this restricted canocc relation which forms the starting point for the method $\mathcal{R}$ described in the last section.

1. The description used did not include every face in the data base provided by Link; there are some 199 faces missing. The reason for the exclusion derives from errors of interpretation which occured in the process of transfering the data base from Link to Stanford. The description which resulted from the removal of the "bad faces" was displayed and inspected visually; its appearance was that of a hilly landscape with normal features. Thus the description, though not identical to the Link data base, is as reasonable a test as any of the effectiveness of our methods.

In the experiments performed so far, the synthesis method has been applied to the landscape L for only one of the ten pie slices. The features of the landscape (mountains and valleys) are oriented in more or less random directions, so there is reason to believe that similar results would be obtained for each of the pie slices. Also, the steeply up and steeply down orientations should yield results which are better, and not worse, than the horizontal orientations.

The synthetic program T_3 produced for priority sorting for the landscape within one slice of viewer directions had the following attributes:

Worst case number of decision nodes encountered during any execution of T_3: 53

Expected number of decision nodes encountered during an execution of T_3, assuming that at each decision node the two possible outcomes of the decision are equally likely: 27

Total number of decision nodes in T_3: 85

For the flight simulation application, one would wish to implement not T_3 itself, but rather a variant of T_3 which takes advantage of the fact that, in flight simulation and similar applications, the position of the viewer changes smoothly from frame to frame. This can be done by arranging to keep a record from each priority computation of the result of that computation, and of also which computation path was taken, that is, a record of which branch was followed at each decision node encountered in the course of the computation. Then, for following frames, the program checks whether the new viewer positions yield new computation paths; as long as the computation path does not change, there is no need to modify the priority sort computed at an earlier frame. Also, as long as the change takes place fairly late in the computation path, only a part of the priority sort will need to be recomputed. In any case, a machine language program of the kind just described can be automatically derived in a straight-forward manner from T_3. The construction of a machine language program has not actually been carried out, nor is there any reason to do so for the present purposes, since the relevant parameters (size and speed) of the machine language program can be easily derived from T_3. Here are those parameters.

Size(estimate) : 1500 36 bit words

Expected number of machine language instructions executed in order to verify that a given priority sort is still valid for a new frame: 8 instructions per plane * 27 planes = 216 instructions

Thus, the total size of the synthetic program, with all orientations accounted for, would be about $1500 * 12 = 18{,}000$ words.

The average number of instructions executed per frame should not be much higher than the expected number of instructions required for verification since, for most new frames, the computation path will not change.

It is difficult to estimate in the general case how the size and speed of the special purpose program produced by our method for an arbitrary scene will depend on the characteristics of that scene. For one thing, it is not only the size of the scene that is relevant, but also the details of its structure. (To see this, note that there are scenes of arbitrary size with the property that one priority ordering works for all positions of the viewer.) However it may be useful if we give some extremely rough estimates based on our experience with synthetic programs derived for fragments of the landscape used for the experiment. It appears that for this kind of scene, both the size and the running time of synthetic programs are given by expressions of the form $k * f(n) * n$, where $f(n)$ is a function which grows very slowly with n; perhaps $f(n) \approx log(n)$. The constant k of

proportionality for the running time (in machine instructions executed) is small - something like .02, whereas for the size (in words of memory), k is something like 10.

5. Comparison to other algorithms for priority sorting

There are two algorithms described in the literature which exploit object coherence in performing priority computations, namely those of Schumacker [Sutherland et al, 1974], and Fuchs[1980]. (Another algorithm which agressively exploits object coherence - though not for priority sorting - is that of Hubschman and Zucker[1981].) These algorithms are similar to each other in that they both make use of a hierarchical decomposition of the scene by means of a collection separating planes; the separating planes are computed off-line. Such a decomposition is referred to as a "binary space partitioning tree" (BSP) by Fuchs. There is a superficial resemblance between the special purpose programs which we produce and the programs which use the BSPs at runtime to perform the priority sort. The resemblance is that in both cases, the method involves computing the viewer's position relative to a set of planes. The resemblance is superficial because the following significant differences exist: (1) The BSP algorithms of Schumacker and Fuchs *traverse* the binary space partitioning tree - comparing the viewer position to the plane at *every* node - rather than *executing* a decision tree (or the like), in which only a fraction of the nodes are visited. (2) The planes which appear at decision nodes in our synthetic programs are not separating planes in any significant sense (although they will separate one *face* from another); in any case, there is not the kind of systematic geometric relationship between the planes and the priority orderings produced that exists for BSP algorithms.

The algorithm of Schumacker was used by Link for dealing with the landscape L on which we performed our experiments. The number of separating planes for L was about 200 (it was not possible to get a precise figure), so that the number of plane comparisons needed for a priority sort in this case is $200/27 = 7.4$ times the expected number required by our synthetic program. Also, the overhead per plane for keeping track of the traversal of a BSP is higher than the overhead per plane for our synthetic progam; at least 12 (rather than 8) instructions per plane are needed. So all in all, the synthetic program is about $1.5 * 7.4 = 11.1$ faster than Schumacker's for this example. This comparison holds for Fuchs' algorithm as well, since the BSP which it would construct would not in any case be smaller than that used for the Schumacker algorithm. However, the space requirements of Schumaker's algorithm are smaller by about a factor of ten than ours. This is not as bad as it sounds, since the space needed to store a synthetic program for priority sorting, though large, is still of the same order as the space needed for storing other data about the scene, such as smooth shading and color information. Another disadvantage of our approach is that the amount of computer time required to construct synthetic programs is quite large. The synthesis of T_3 (which deals with only one of twelve orientations) took about one hour of cpu time on a PDP-10/KL-10 computer. Still, this is not prohibitive if the scene for which the synthesis is being carried out is to be used for many simulations.

The main advantage of our method over Schumacker's is not speed, but flexibility. Although our trial landscape L is separable into clusters in the way required by Schumacker's algorithm, we make no use whatever of that fact; our results would be the similar for another landscape of the same general kind for which no set of separating planes existed. The restriction to separable scenes needed for Schumacker's algorithm is a serious one in pratical terms; designers of data bases intended for use with Schumacker's algorithm need to expend substantial amounts of effort to assure that the separability criterion is met. The algorithm of Fuchs does not impose this restriction, but it has another kind of problem. Namely, as the size of a scene grows, the number of nodes in the BSP can increase very rapidly. In Fuchs[1981], an upper bound on the number of

nodes is given which is cubic in the number of faces. If this bound is approached for the scene L, then Fuchs' algorithm is not usable. Of course, there is no reason to expect *a priori* that the upper bound is a relevant estimate for situations which arise in practice. For the considerably smaller examples which Fuch's considered, the number of nodes in the BSP did not grow at a rapid rate. So, without further data, it is difficult to estimate exactly what the result of applying Fuchs' algorithm to the landscape L would be.

6. Related work on program synthesis and manipulation

As indicated earlier, the techniques which we have used for producing and optimizing decision trees are adapted from standard methods which have been employed in many fields, including program transformation [Burstall and Darlington, 1977], planning [Sproul 1977], program synthesis of the traditional kind [Manna and Waldinger, 1980] and program specialization [Beckeman et al 1976; Emmanuelson, 1980]. The work reported here differs from work in all of these fields in more or less the same way, namely, in that we treat classes of problems defined in a mathematical rather than universal style. For example, in program specialization, one looks for methods which can be usefully applied to any program in a given language; thus, the goal of the enterprise is the development of *universal* methods which will apply to all problems formulated in a particular way. Another example is that of "knowledge based" progam synthesis of the kind developed by Barstow[1977]. The aims are again universal, both in the sense that the class of problems to be treated, although restricted to the program's "domain of expertise", is still much wider than the class which we treat, and in the sense that the mechanism for synthesis is intended to have universal application; only the "knowledge" encoded in a set of production rules is particular to the domain of expertise. In contrast, the results which we obtained for the hidden surface elimination problem depended on adapting our methods specifically to the problem at hand.

There are many algorithms from the mainstream of computer science which may be seen as synthesizers of special purpose programs. For example, there is a wide and useful class of algorithms for recognition of patterns in strings [Aho, Hopcroft, and Ullman, 1974] which, when given the pattern, proceed by constructing a finite automaton which in turn performs the search. A finite automaton, like a decision tree, is a simple variety of program, and in this sense, these algorithms construct programs for the special purpose of finding an instance of a pattern in a string. The hidden surface elimination algorithm described in section 3 should be regarded as a new piece of work in this tradition.

References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D.[1974], *The design and analysis of computer algorithms,*Addison-Wesley, Reading Mass., 1974, see Chapter 9: pp. 317-361

Barstow, D.[1977], *A knowledge based system for automatic program construction,*Fifth International Joint Conference on Artificial Intelligence, Cambridge, August, 1977

Beckeman,L., Haraldsson, A., Oskarsson,O., and Sandewall, E.[1976], *A partial evaluator and its use as a programming tool,*Artificial Intelligence Journal 7,1976, pp. 319-357

Burstall R.M., and Darlington, J.[1977], *A transformation system for developing recursive programs,*JACM, Vol. 24, No. 1, January 1977

Emmanuelson, P.[1980], *Performance enhancement in a well-structured pattern matcher through partial evaluation,*Ph.D. Thesis, Software Systems Research Center, Linköping University, Linköping, Sweden, 1980

Fuchs, H., Kedem, Z.M.,and Naylor, B.F.[1980], *On visible surface generation by a priori tree structures,*Computer Graphics, Vol. 14, No. 3, July 1980,p. 124

Hubschman, H., and Zucker, S.W.[1981], *Frame-to-frame coherence and the hidden surface computation: constraints for a convex world,*Computer Graphics, Vol. 15, No. 3, August 1981,p. 45

Knuth, D.E.[1968], *The art of computer programming, vol 1: Fundamental algorithms,*Addison-Wesley, Reading Mass., 1968, pp. 258-268

Kreisel, G.[1981], *Neglected possibilities for processing assertions and proofs mechanically: choice of problems and data,*in: P. Suppes [ed.], University-level computer-assisted instruction at Stanford: 1968-1980. Stanford Calif.; Stanford University, Institute for Mathematical Studies in the Social Sciences, 1981

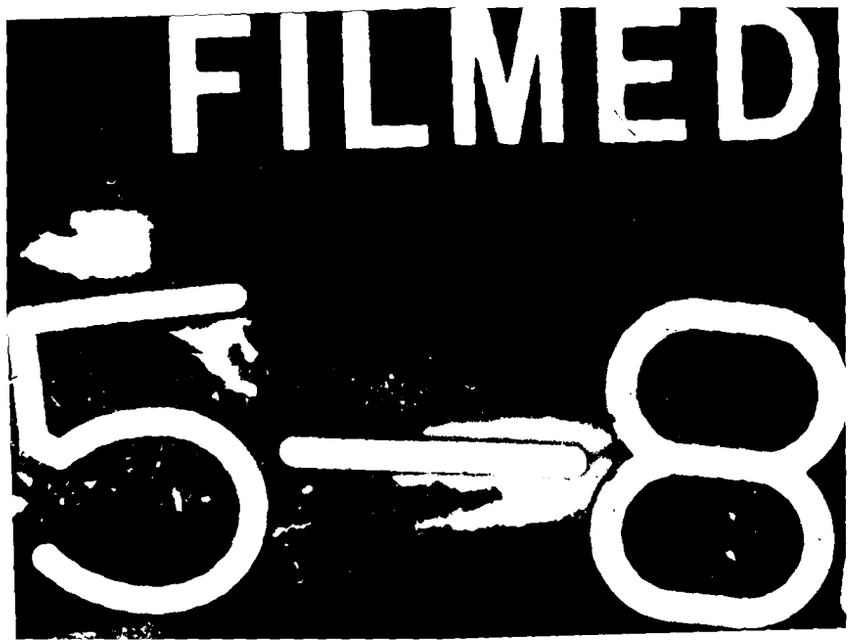
Manna, Z., and Waldinger, R.[1980], *A deductive approach to program synthesis,*ACM Transactions on programming languages and systems, Vol. 2, No 1., January 1980

Newell, M.E., Newell, R.G., and Sancha, T.L.[1972], *A new approach to the shaded picture problem,*Proc. ACM National Conference, 1972

Robson, J.[1979], *The height of binary search trees,*The Australian Computer Journal, 11(1979), pp 151-153

Sproull, R.F.[1977], *Strategy construction using a synthesis of heuristic and decision-theoretic methods,*Xerox PARC technical report CSL-77-2, July, 1977

Sutherland, I.E.,Sproull, R.F., and Schumacker, R.A.[1974], *A characterization of ten hidden-surface algorithms,*Computing Surveys, Vol. 6, No. 1, March 1974


FILMED
5-8